数値表現

数値表現形式：**ss'fnn...n**

ss は，定数の**ビット幅**を 10 進数で表します

f は，**基数**を表します

b が 2 進，o が 8 進，d が 10 進，h が 16 進

nn...n は，**定数値**を表します。各基数で許される値を書くこ

Verilog	ビット幅	基数	2進表現
1'b0	1	2進	0
4'b0100	4	2進	0100
4'd4	4	10進	0100
4'd7	4	10進	0111
8'hf0	8	16進	11110000
8'd1000_1111	8	2進	10001111
1	32	10進	000...0001

モジュール 回路動作を記述する

Verilog-HDLでは、回路動作は
キーワードmoduleとendmodule
で囲まれたモジュール内に定義します。

キーワードmoduleに続き、モジュール名と、
FPGAの入出力ポート・リストを記述します。

モジュール名は　　**PwmCtrl**
入出力ポートリストは**RST_N,CLK,LED0**

3つの入出力ポートを持つことがわかります。

モジュール構文：
module モジュール名 (信号名　{, 信号名});
　{ポート宣言文}
　{パラメータ宣言文}
　{内部信号宣言文}

　{順序的構文}
　{プリミティブ}
　{階層構造}

endmodule

```
//PwmCtrl.v
//
module PwmCtrl (  RST_N,   CLK,  LED0 );

input CLK, RST_N;
output LED0;
reg [27:0] counter0;

always@( negedge RST_N or posedge CLK )
begin
   if (RST_N == 1'b0 ) begin
      counter0 <= 0;
   end else begin
      counter0 <= counter + 1;
   end
end

assign LED0 = counter0[27];
endmodule
```

ポート宣言文

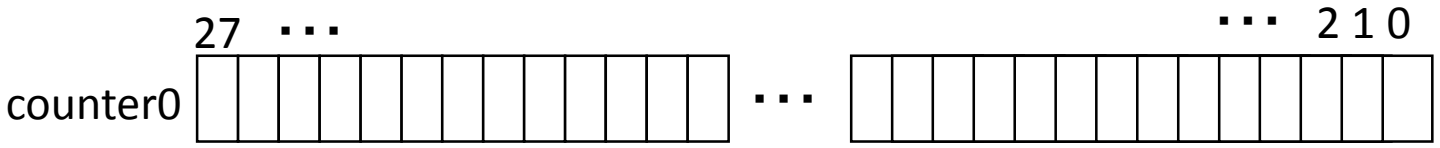
- ポート信号は、ポート宣言文によって入力(input)、出力(output)、双方向入出力(inout)のいずれかのモードを指定し、ベクタの場合はベクタ幅を指定します。

```
input       CLK, RST_N;
output      LED0;
reg [27:0]  counter0;
```

入力，出力，レジスタの宣言

regはデータタイプです。

counter0　という名前の28bit幅のレジスタ変数を用意します。




```
//PwmCtrl.v
//
module PwmCtrl (  RST_N,   CLK,  LED0 );

input CLK, RST_N;
output LED0;
reg [27:0] counter0;

always@( negedge RST_N or posedge CLK )
begin
   if (RST_N == 1'b0 ) begin
      counter0 <= 0;
   end else begin
      counter0 <= counter + 1;
   end
end

assign LED0 = counter0[27];
endmodule
```

データタイプ

データタイプは、信号や変数の持ちうる状態を指定したもの

レジスタ宣言 reg：データを保持する記憶素子

一度代入されてから次に代入されるまで値を保ちます。レジスタへの代入文は、initial文かalways文、またはサブプログラム内の手続き的ブロックでのみ可能です。

ネット宣言 wire:モジュールやゲート同士を接続している物理的な状態

常にゲートや継続代入文によってドライブされ、ドライブされていない時にはハイインピーダンス状態になります。

レジスタ宣言：

reg <[MSB:LSB]> レジスタ名；

ネット宣言：

ネットタイプ <[MSB:LSB]> <電荷ストレングス> <遅延> ネット名；
ネットタイプ := wire|wand|wor|tri|triand|trior|tri1|tri0|trireg|supply0|supply1
電荷ストレングス := small | medium | large

代入文

代入文には、2種類の基本的な形式があります。

- 継続的代入文　　ネットに値を継続的に代入する
  ネット型変数への代入には、assign文でのみ可能
- 手続き的代入文　　手続きブロックの中でレジスタに値を代入する
  レジスタ変数の信号への代入は,always文、initial文、task、functionの中で可能です。

if-else文、case文、ループ文、遅延制御文などの動作構文

FPGAでは、ノイマン型のCPUとは一命令ずつ処理を行いません。
電気回路なので同時に動作します。
そのため、HDL言語も、C言語のように一行ずつ処理を書くプログラムと異なり部分があります。
代入文がFPGAの特徴をよく現しています。
FPGAでは物理的な電気結線でFPGA内に論理回路を構築します。
そのため、複数の代入文が同時に処理されます。

継続的代入文

- ベクタまたはスカラ・ネットに対して継続的に値をドライブする
- 継続代入文では、右辺に含まれる信号の値が変化すると、自動的に式が評価され左辺へ代入
- ネットタイプの宣言と同時に代入式を定義するか、またはすでに宣言されたネットに対してassign文によって代入式を定義します。
- 遅延値が指定されている場合は、指定された時間だけ代入が先送りされます。

継続代入文構文:

```
wire   <[MSB:LSB]> <遅延> 代入文 ;
assign <遅延> 代入文 ;
```

組合せ回路（assign 文）

assign ネット型変数 = 論理式など;

- 簡単な組合せ回路を記述するときには assign 文を用います。
- assgin 文の左辺は，ネット型変数（ポート宣言，あるいは wire 宣言された変数）でなければなりません。

＜記述例＞

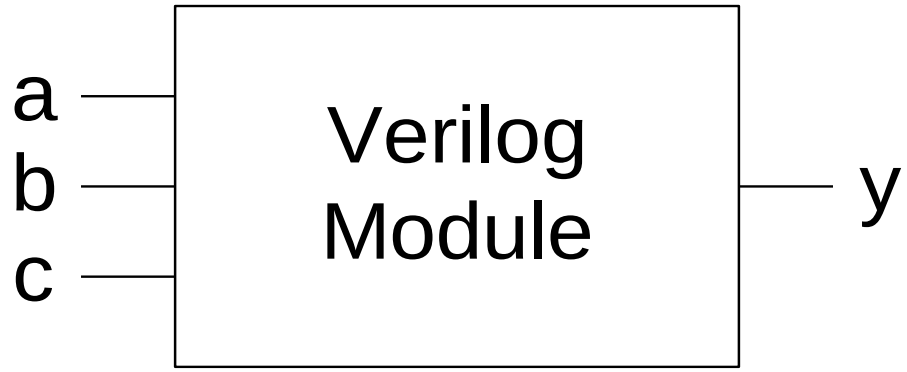
```
wire a, b, c, d;
wire [3:0] s, t, u;

assign c = a | b;                 // 2入力OR
assign u = s & t;                 // 4ビット2入力AND
assign c = (d == 1)? a: b;        // セレクタ（d が1のとき a を，1
                                  //  でないとき b を，c に出力）
```

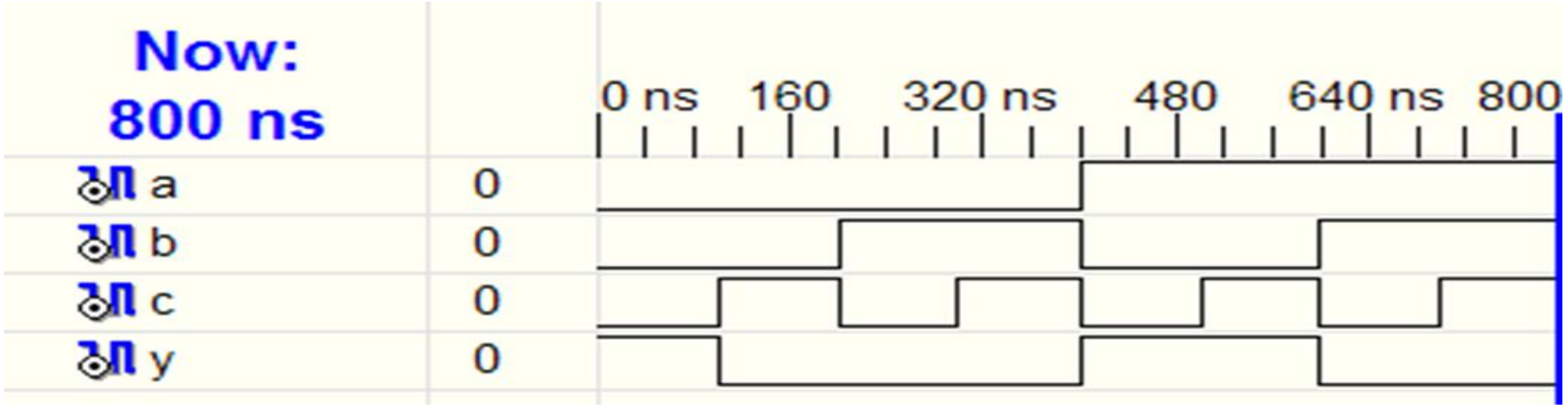
Verilog:

2. ソースを書く　:継続的代入文

```
module example( a, b, c, y);
input a, b, c;
output y;
```

```
module example(input  a, b, c, output y);
assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```



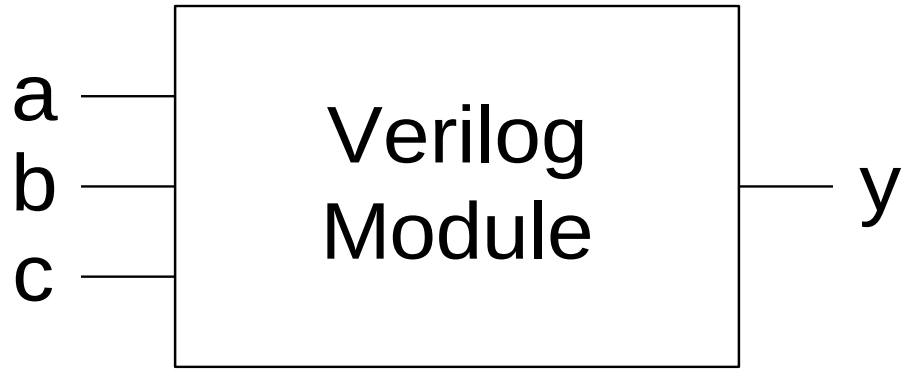


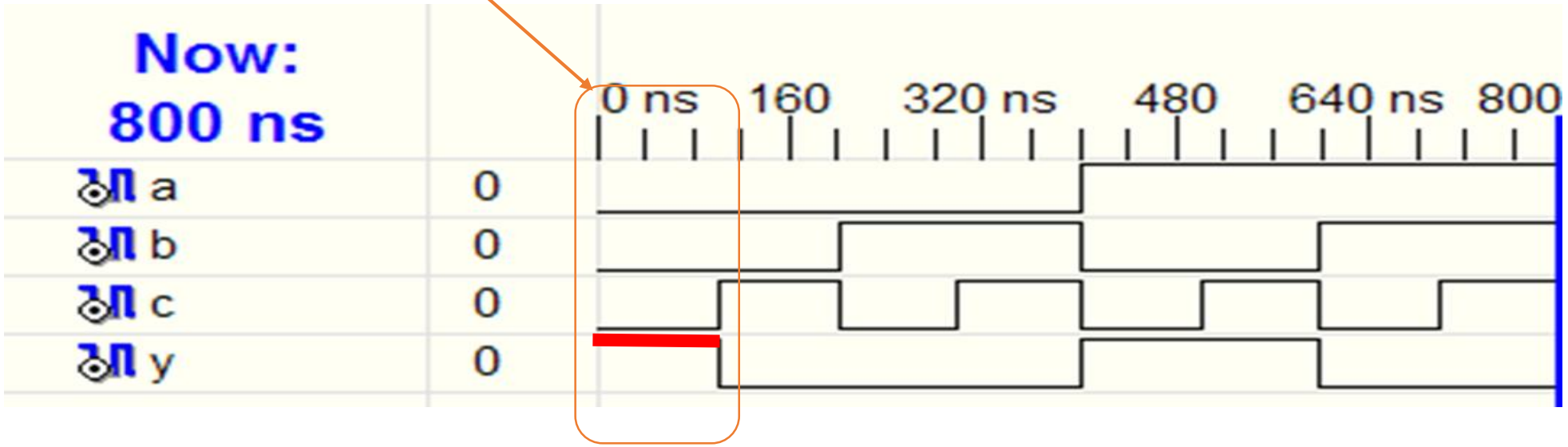
Verilog:

2. ソースを書く　:継続的代入文

```
module example( a, b, c, y);
input a, b, c;
output y;
```

```
module example(input  a, b, c, output y);
assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```



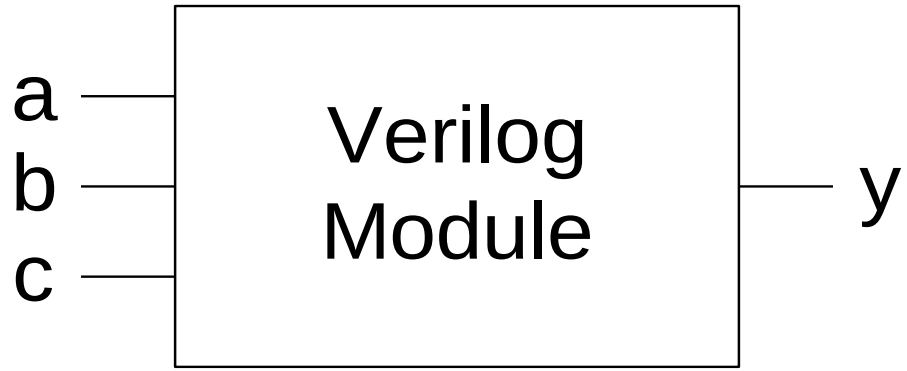


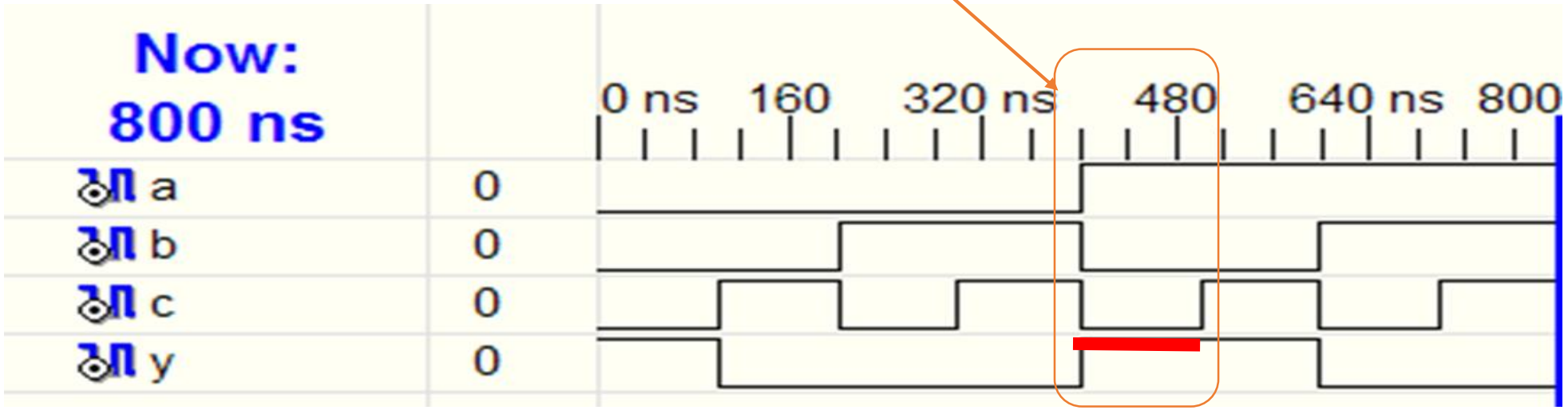
Verilog:

2. ソースを書く　:継続的代入文

```
module example( a, b, c, y);
input a, b, c;
output y;
```

```
module example(input  a, b, c, output y);
assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```



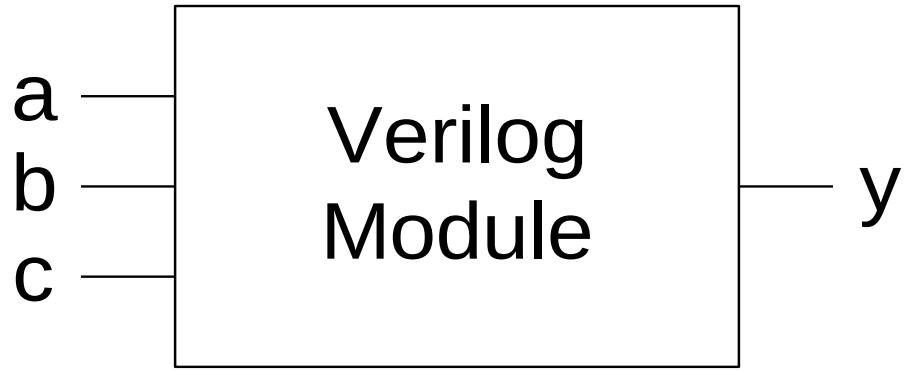


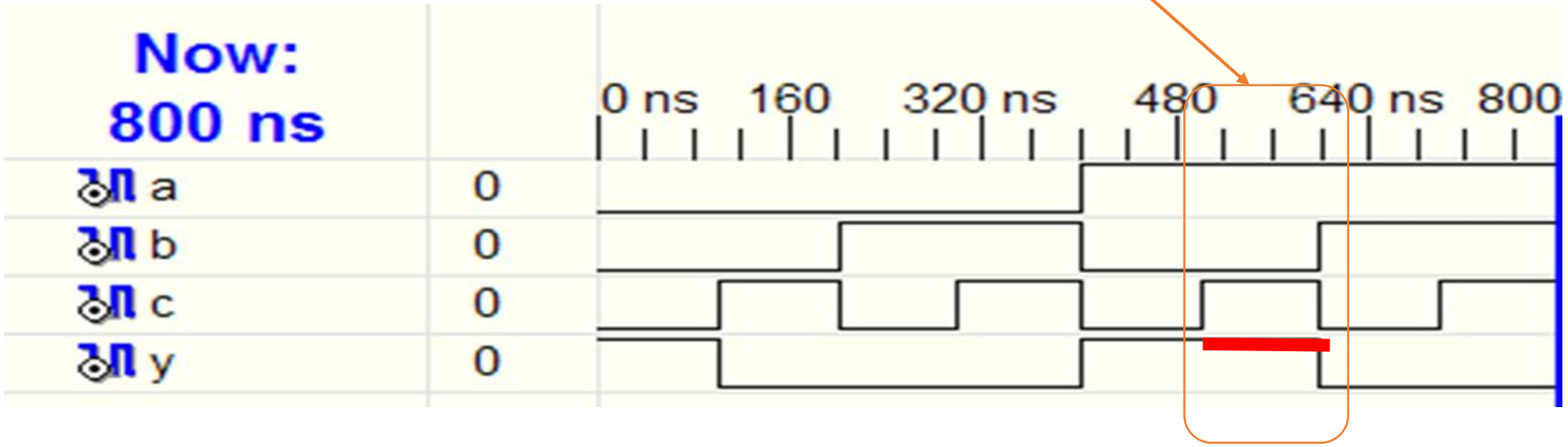
Verilog:

2. ソースを書く　:継続的代入文

```
module example( a, b, c, y);
input a, b, c;
output y;
```

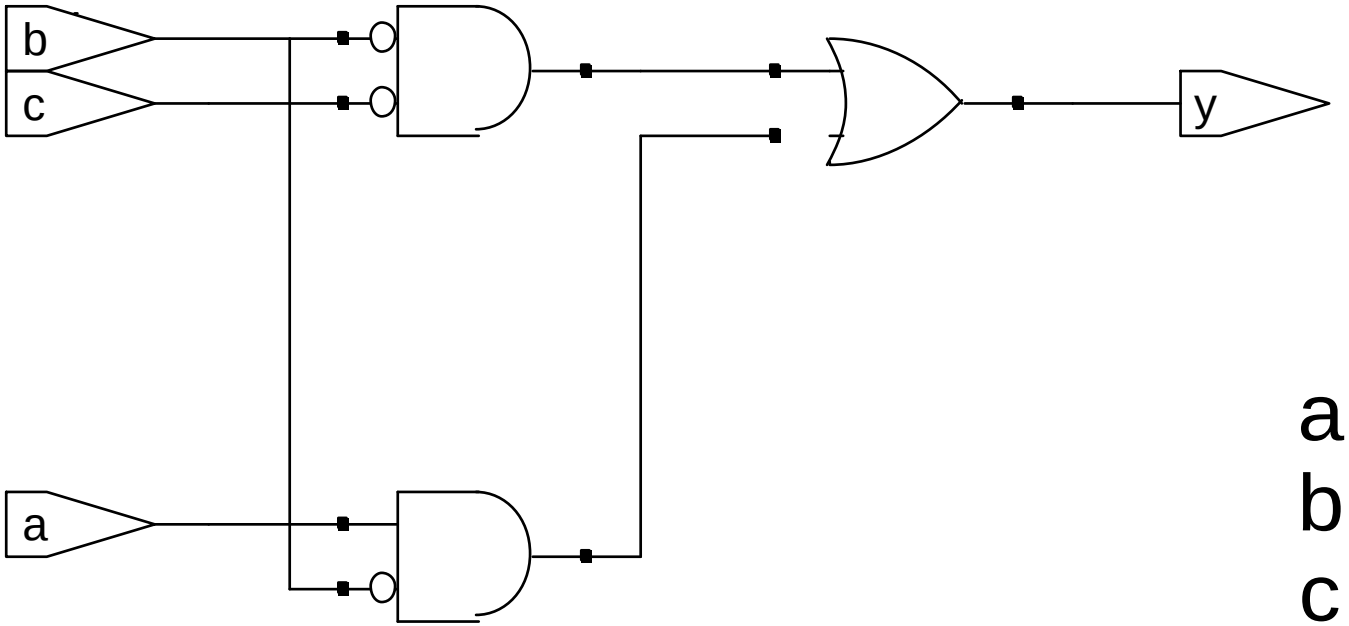
```
module example(input  a, b, c, output y);
assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

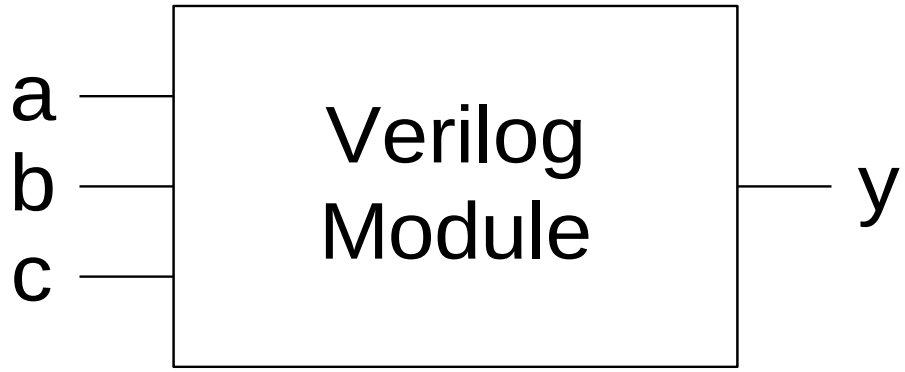




Verilog:

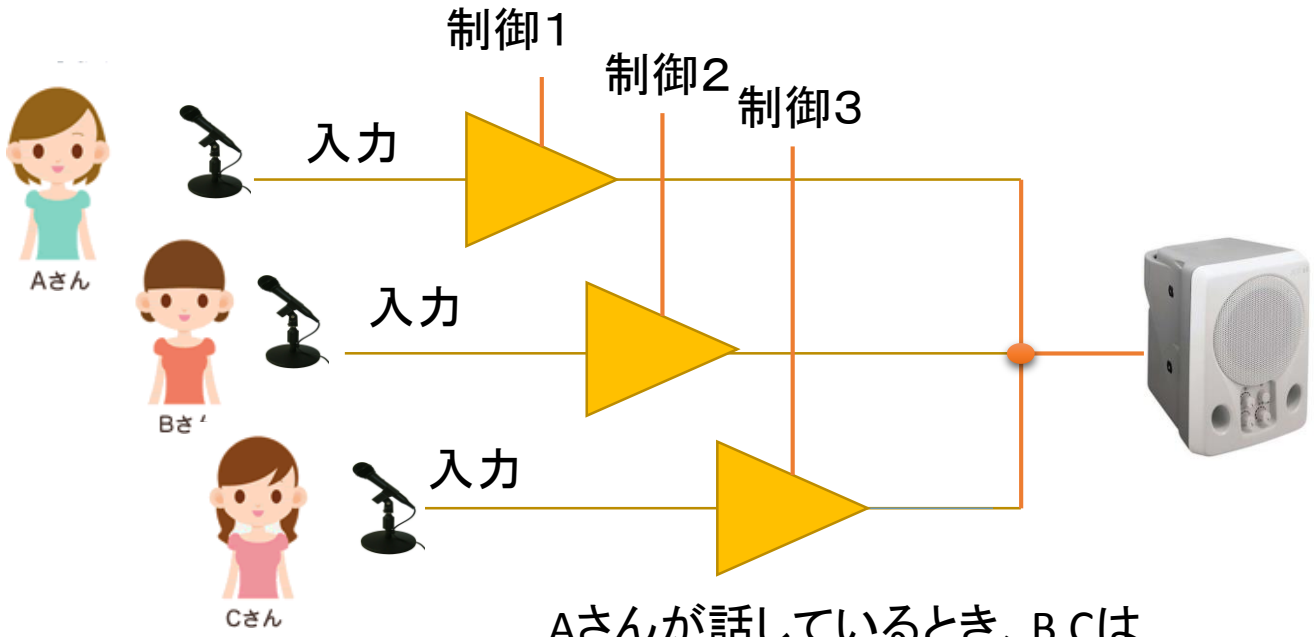
```
module example(input  a, b, c, output y);
assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```



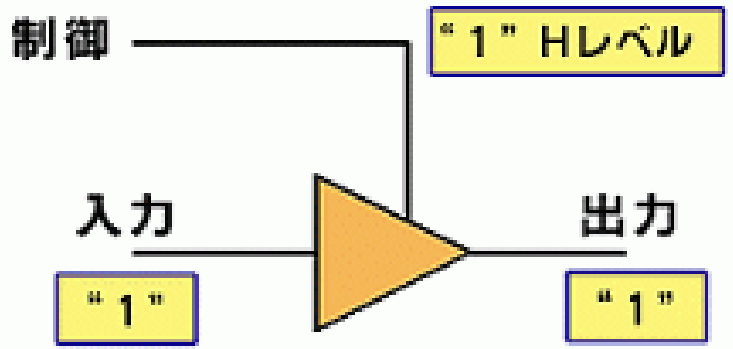


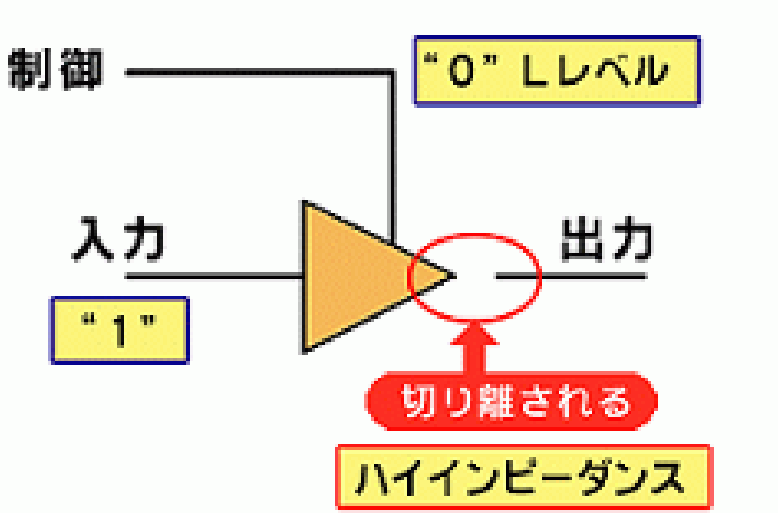
3ステート　ハイインピーダンス

入力信号により、出力の信号が決まり、“1”または“0”の状態となります。しかし、一部の論理素子ではこの“1” “0”以外に、もう1つの状態を持つものがあります。
ハイインピーダンス状態と呼びます。



Aさんが話しているとき、B,Cは
O（OV=GND) を出力してはいけない

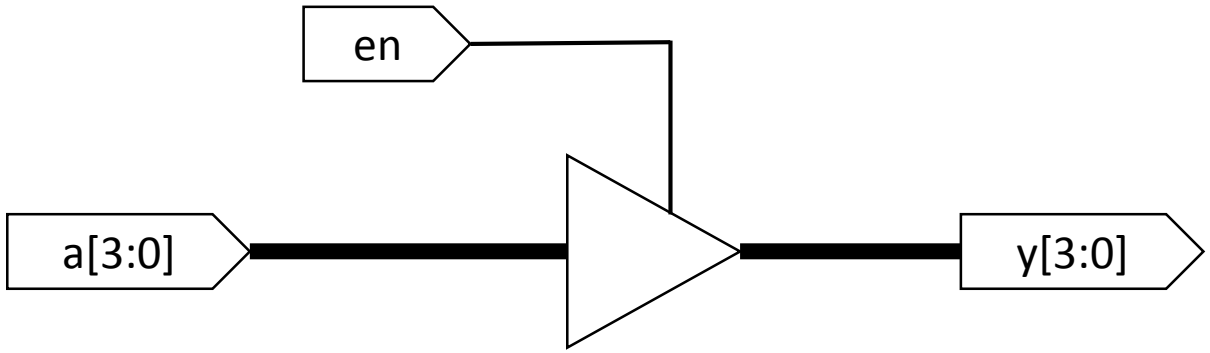




ハイインピーダンスアウトプット

```
module tristate(input  [3:0] a,
                input        en,
                output [3:0] y);
   assign y = en ? a : 4'bz;
endmodule
```

■条件演算子(Conditional Operator)
式1 ? 式2 : 式3
式1 が 0 でない(真である)場合には,
式2 を返し, 0(偽)のときには 式3 を
返します。



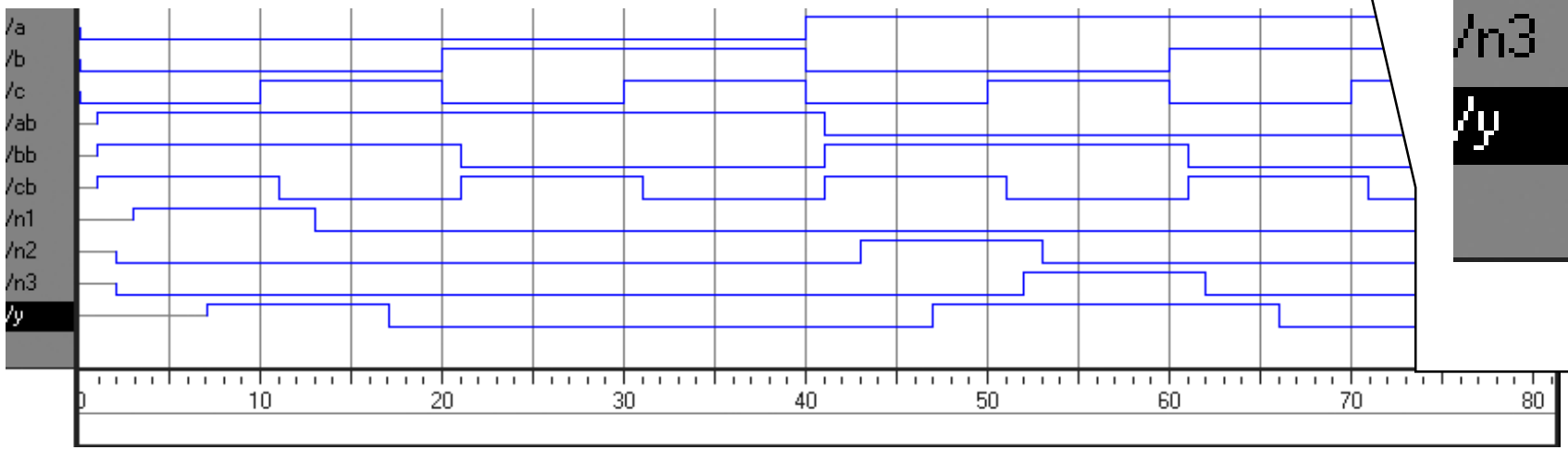
入力制御信号　en が1のときは、
yにはaの値がそのまま出力される。

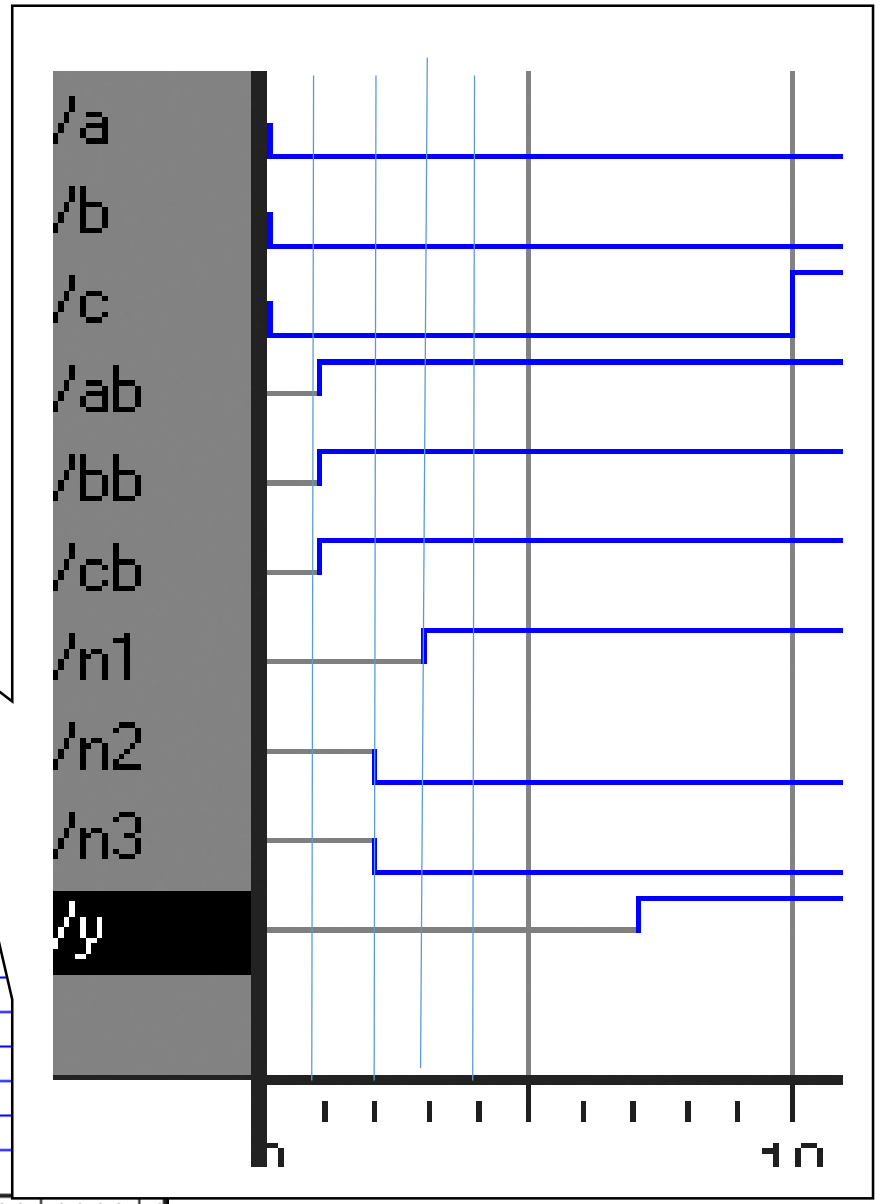
入力制御信号　en が0のときは、
yはハイインピーダンス状態となる。

遅延値が指定されている場合

#遅延値

```
module example(input  a, b, c,
               output y);
  wire ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```





遅延制御

#遅延式

遅延値は整数で、記述には時間単位(ns、psなど)は指定せず、遅延値は時間単位を持たないユニット時間となります。遅延値がどのような時間単位であるかは、`timescale文によって指定 します。たとえば、遅延値10が10 nsを意味させるためには、

`timescale 1 ns / 1ns

のように指定します。

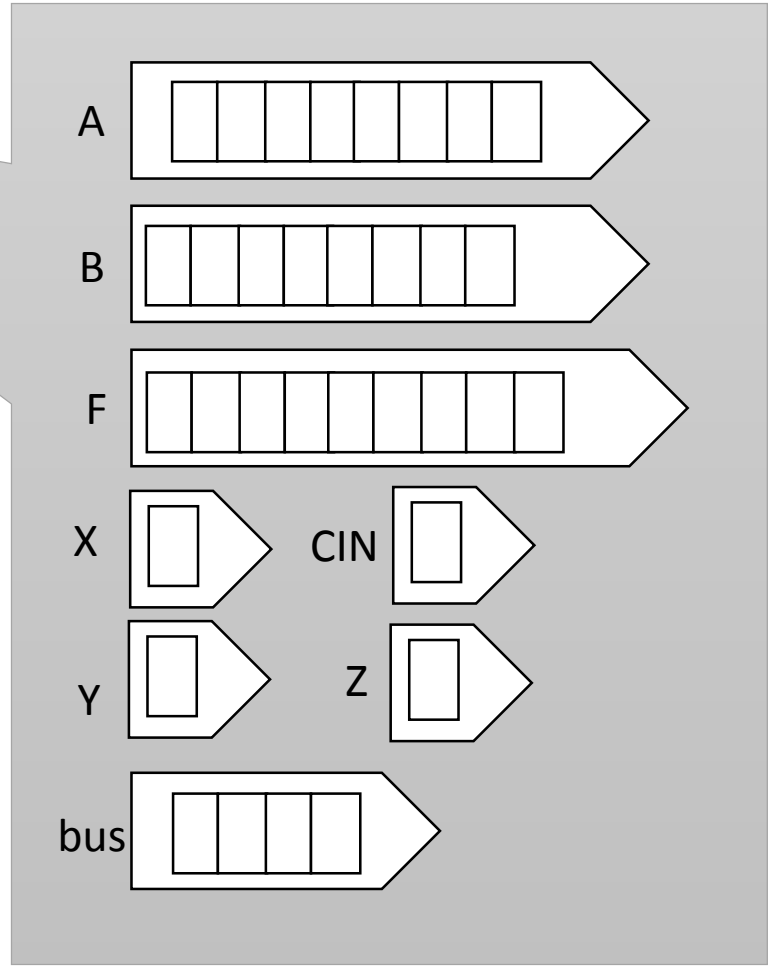
1ユニットを100psにし、丸め精度を100psにするには、

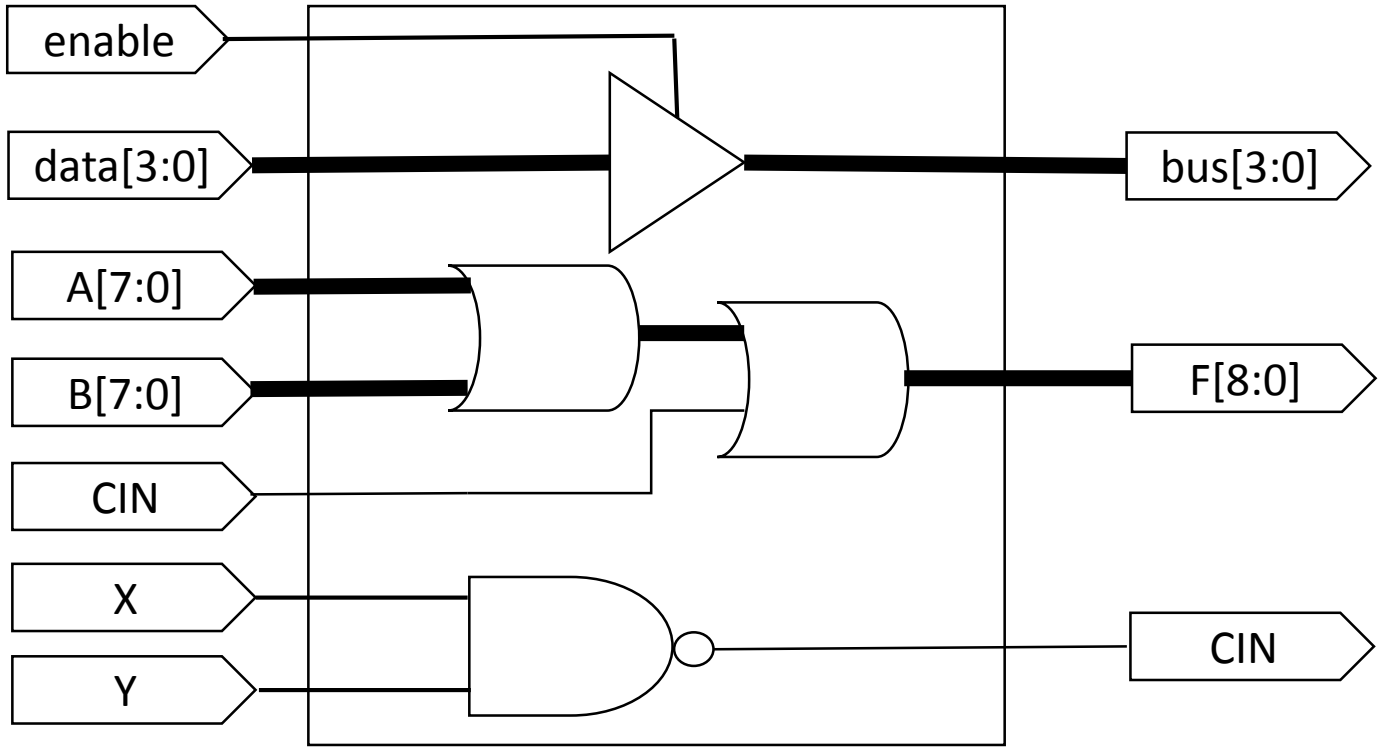
`timescale 100ps/100ps

`timescale
シミュレーションの単位を設定する。
(書式) `timescale 1ユニットの時間/丸めの精度

2. ソースを書く　:継続的代入文

```
WIRE [7:0] A, B;
WIRE [8:0] F;
WIRE CIN, X, Y, Z;
WIRE [1:4] BUS ;

ASSIGN BUS = ENABLE ? DATA : 4'BZ;
ASSIGN #1 F = A + B + CIN;
ASSIGN Z = ~(X & Y);
```





手続き的代入文

手続き代入文は、値をレジスタ(変数)に代入するもので、always、initial、task、functionのような手続き文内で使用します。

手続きブロックは、initial文またはalways文で始まる

initial文は1回だけ実行

```
initial begin
    A = 0;
    #10 B = 1;
    #10 A = 1;
end
```

always文は繰り返して実行: (順序回路をしめす)

always文に引き続きセンシティビティ・リストを定義

手続きブロック内に遅延制御文(#)

手続きブロック内にイベント制御文(@)

イベント制御では、posedge　: 立ち上がりエッジ

negedge　: 立ち下がりエッジ

or文を用いてイベントの活性化

```
always begin
    C = 1;
    #10 C = 0;
    #10;
end
```

```
always @(A or B)
    D <= A or B;
```

ノンブロッキング代入(＝)

=を用いた手続き的代入文は、式が評価すると同時に代入を実行

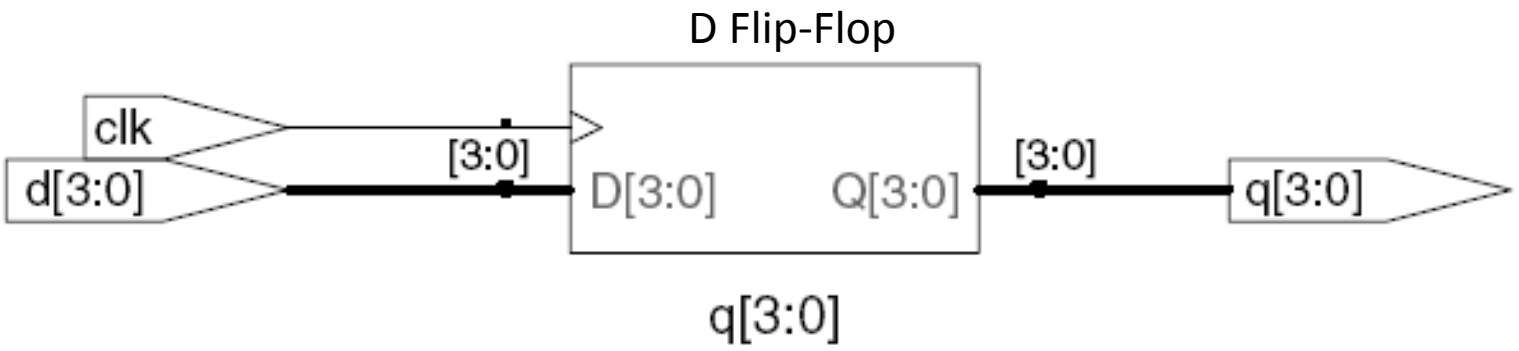
```
always @(posedge CLK)
    Q <= #1 DATA;
```

ブロッキング代入(＜＝)

右辺の評価が完了しても、代入はその時刻での他の代入文の評価が完了するまで先延ばし

手続き文 always

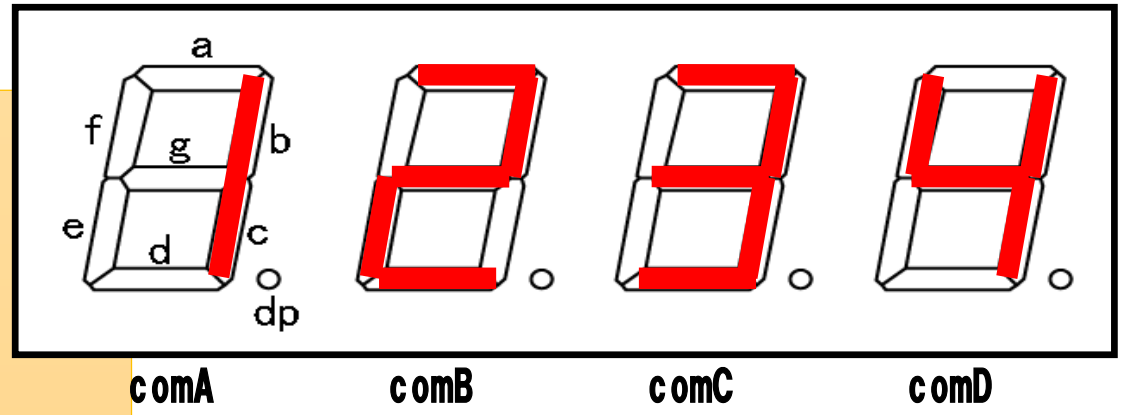

```
module flop(input             clk,
            input       [3:0] d,
            output reg  [3:0] q);

  always @ (posedge clk)
    q <= d;
endmodule
```



clk　のたち上がりで
qの値を更新します。(qにはdの値が入る)

手続き文 always と case の組合せ

```
module sevenseg(input      [3:0] data,
                output reg [6:0] segments);
  always @(*)
    case (data)
      //                 abc_defg
      0: segments = 7'b111_1110;
      1: segments = 7'b011_0000;
      2: segments = 7'b110_1101;
      3: segments = 7'b111_1001;
      4: segments = 7'b011_0011;
      5: segments = 7'b101_1011;
      6: segments = 7'b101_1111;
      7: segments = 7'b111_0000;
      8: segments = 7'b111_1111;
      9: segments = 7'b111_1011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```



ブロッキング代入文とノンブロッキング代入文

Verilog-HDL では並列性のある命令の流れを記述することができます。
それを実現するのがノンブロッキング代入文です。
通常のプログラム言語のように普通の代入文をブロッキング代入文と呼びます。

ブロッキング代入文が「代入処理が終了して次の文を実行する」

ブロッキング代入文	ノンブロッキング代入文

initial
begin
a = 3;
b = 5;
 	
#10
a = b + 1;
b = a + 1;
end
	initial
	begin
	a <= 3;
	b <= 5;
	
	#10
	a <= b + 1;
	b <= a + 1;
	end

#10 は，10 単位時間後

同時刻に行われている。

ブロッキング代入文	ノンブロッキング代入文
a = 6, b = 7	a = 6, b = 4

ノンブロッキング代入文では「同時刻の代入文は，右辺を評価してから代入する

ブロック文

ブロック文は、2つ以上の手続き文をまとめて、1つの文のように定義するものです

- 順序ブロック:　beginとendで区切る
- 並行ブロック:　forkとjoinで区切る

並行ブロック構文:
```
fork <: ブロック名>
    {パラメータ宣言}
    {内部データタイプ宣言}

    {手続き文}
join
```

順序ブロック構文:
```
begin <: ブロック名>
    {パラメータ宣言}
    {内部データタイプ宣言}

    {手続き文}
end
```

順序ブロック

```
always @(negedge RST_N or posedge CLK)
begin
        if (RST_N == 1'b0) begin
                counter0 <= 0;
        end else begin
                counter0 <= counter0 + 1;
        end
end
```

RST_N の立下りと
CLK　の立上りで実行される

ブロッキング代入
count0にOを入れる

ブロッキング代入
count0＋1したのを
count0に代入

例　A=4'b0011：　4ビットの 0011 = 10進数の3
A =2'b01 : 2ビットの01　＝10進数の1

組合せ回路(assign 文)

assign ネット型変数 = 論理式など;

- 簡単な組合せ回路を記述するときには assign 文を用います。
- assgin 文の左辺は，ネット型変数(ポート宣言，あるいは wire 宣言された変数)でなければなりません。

＜記述例＞

```
wire a, b, c, d;
wire [3:0] s, t, u;

assign c = a | b;              // 2入力OR
assign u = s & t;              // 4ビット2入力AND
assign c = (d == 1)? a: b;     // セレクタ(d が1のとき a を，1
                               //  でないとき b を，c に出力)
```

```
module PwmCtrl (
        RST_N,
        CLK,
        LED0
);

input       CLK, RST_N;
output      LED0;
reg [27:0]  counter0;

always @(negedge RST_N or posedge CLK)
begin
        if (RST_N == 1'b0) begin
                counter0 <= 0;
        end else begin
                counter0 <= counter0 + 1;
        end
end

assign LED0 = counter0[27];

endmodule
```

FPGAの入出力ポート

入力，出力，レジスタの宣言

RST_Nの立ち下がりとCLKの立ち上がりで動く

RST_Nが0のときカウンタはクリアされ，それ以外（CLKの立ち上がり）ではカウント・アップされる

カウンタの最上位ビットをLED0につなぐ